IMPROVING KUBERNETES SERVICE AVAILABILITY THROUGH CHAOS

September 2022

AUTHOR(S):
Nivedita Prasad

SUPERVISOR(S):
Ricardo Rocha
Spyridon Trigazis

CERN openlab

ABSTRACT

Chaos Engineering is the process of testing a distributed computing system to ensure that it can withstand unexpected disruptions. It relies on concepts underlying chaos theory, which focus on random and unpredictable behavior. The goal of chaos engineering is to identify weakness in a system through controlled experiments that introduce random and unpredictable behavior.

There are many tools available, with different level of maturity. Tools like:

- Chaos Mesh
- Litmus Chaos
- Chaos Toolkit
- Pumba

This project aims at selecting and integrating one of these tools into the CERN Kubernetes offering, with the goal of giving service managers a tool that will dramatically increase confidence on their own service availability.

TABLE OF CONTENTS

1 INTRODUCTION 3

2 What is Chaos Engineering? 3

3 What is Chaos Mesh? 4

4 Architecture 5

5 How does Chaos Mesh work? 6

6 How do we use Chaos Mesh? 6
- 6.1 Installation . . . 7
- 6.2 Chaos Mesh Experiments . . . 7
  - 6.2.1 Stress CPU . . . 7
  - 6.2.2 Stress Memory . . . 8
  - 6.2.3 Network Delay . . . 9
  - 6.2.4 Workflow . . . 10

7 Daemonset 11

8 Conclusions 11

1 INTRODUCTION

During the development process different challenges and difficulties may arise, many unforseen. Detecting and handling those scenarios early is far better than once the product is in production, where the cost of change is much higher.

As an example Facebook went down for 6 hours on Oct 5th, 2021 including WhatsApp, Instagram, and Messenger. The cost associated with that outage is estimated at $160 million. (source)

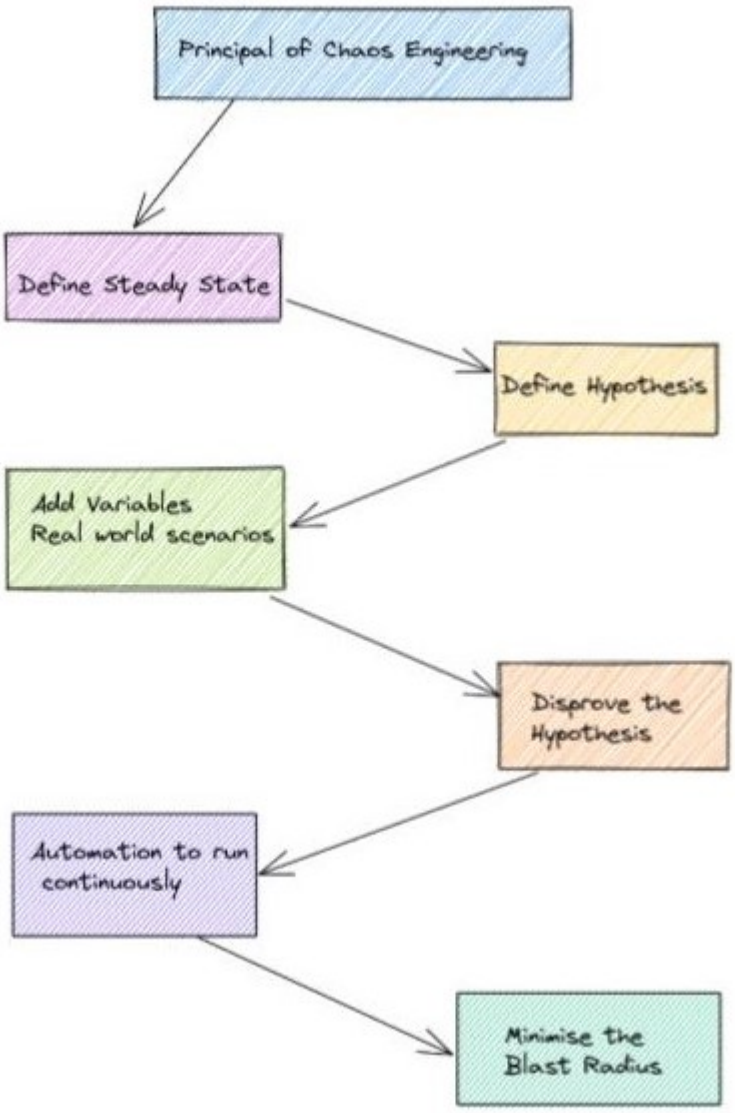
Chaos Engineering tries to create failure scenarios that will reduce the number of similar events happening in production.

2 What is Chaos Engineering?

Chaos Engineering is the discipline of experimenting with a system in order to build confidence in the system's capability to withstand turbulent conditions in production. Or, simply, we intentionally inject fault in the system to check whether the system is resilient or not. The harder it is to disrupt the steady state, the more confidence we have in the behaviour of the system. If a weakness is uncovered, we now have a target for improvement before that behaviour manifests in the system at large.

From principalofchaos.org we have a four-step experimentation process:

1. Start by defining 'steady state' as some measurable output of a system that indicates normal behaviour.
2. Hypothetize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real-world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.
4. Try to disprove the hypothesis by looking for a difference in the steady state between the control group and experimental group.



Typically in Chaos engineering we tend to create an experiment or workflow that involves a hypothesis, such as "latency will not increase despite the loss of n pods", an experiment (deleting pods, slowing the traffic etc.), conducting the experiment, minimising the blast radius and then making the process continuous.
Over the years, there have been many tools created for this practice which are now becoming mature enough to do chaos engineering in production that allow us to predict the effects of failures before they actually happen at scale.

After an initial step comparing possible open source tools in this area we selected **Chaos Mesh** for our implementation.

3 What is Chaos Mesh?

Chaos Mesh is an open-source cloud-native Chaos Engineering platform that orchestrates chaos in Kubernetes environments. Chaos Mesh includes a fault injection method for complex systems on Kubernetes and covers faults in Pods, the network, the file system and even the kernel. It can perform chaos experiments in production environments without modifying the develop-

ment logic of the application.

It is built on Kubernetes CRDs (Custom Resource Definition) to manage different chaos experiments, categorized into three types:

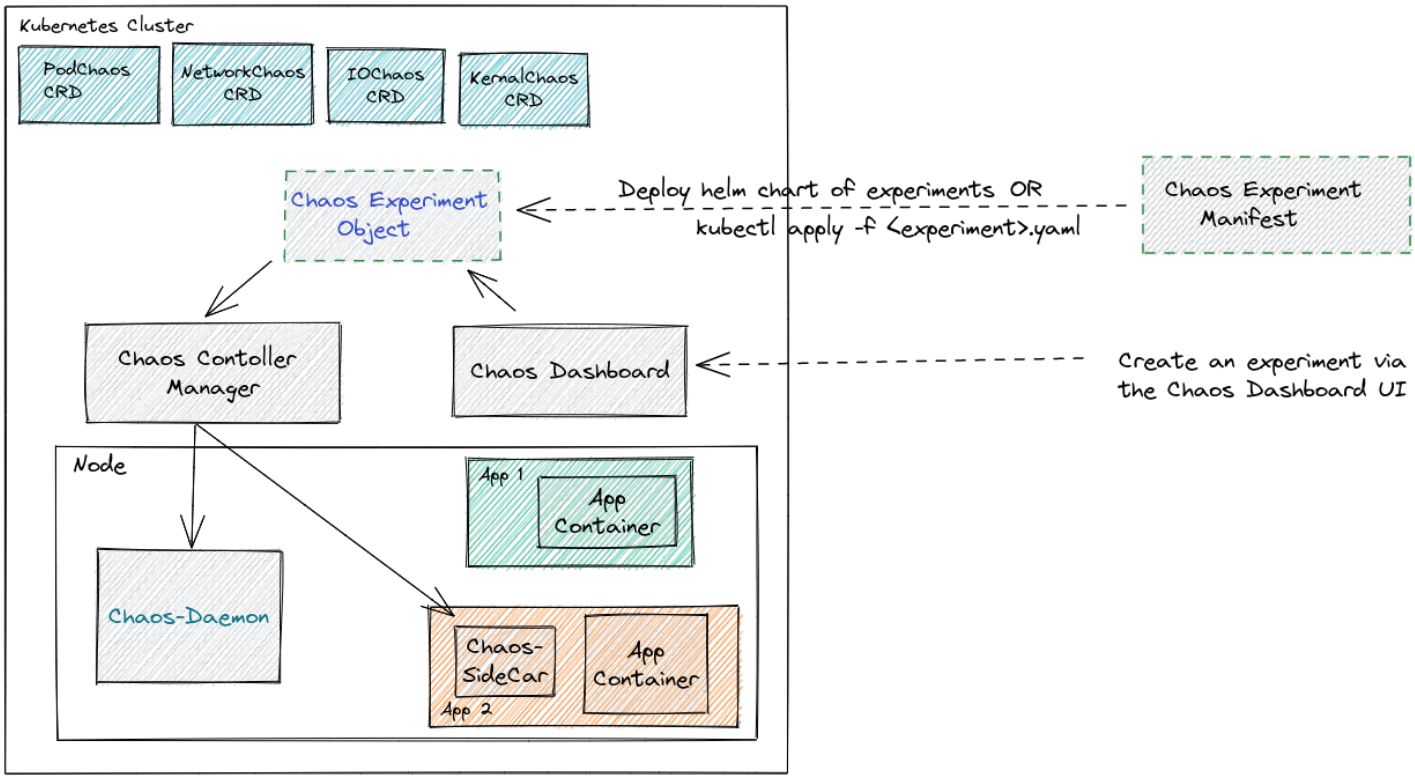
1. Basic Resource types
2. Platform faults
3. Application layer faults

It also provides a visualization dashboard supporting the design of chaos scenarios and monitoring of the status of the chaos experiments.

4 Architecture

There are three major components:

1. **Chaos Dashboard** : Manage and Monitor Experiments.
2. **Controller Manager** : Schedule and Control, Workflow Engine.
3. **Chaos Daemon**: Executive Component, Has Privileged permissions, Mainly interact with network devices, file systems, kernels by hacking into the target Pod Namespace.

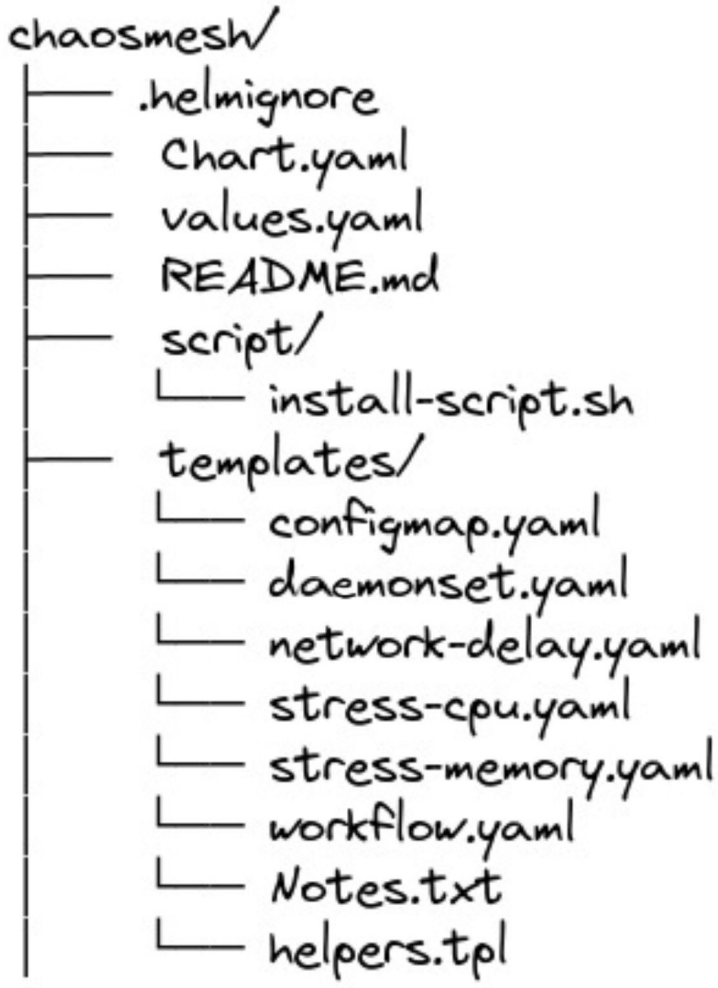


5 How does Chaos Mesh work?

- Chaos Mesh heavily relies on custom resource definitions (CRD) and the principle of reconciliation that all Kubernetes resources follow.
- There are three main ways to interact with the API server via the dashboard, with the kubectl command line client, or using a client API.
- Experiments will first interact with the API server which delegates to the controller manager the management of the resource, including deletion, creation or update of the events.
- Finally these resources are taken by chaos operator daemon which is primarily responsible for accepting commands from the chaos controller manager. This component runs on every node and requires privileges to interact with network devices, file systems and kernels by hacking into the target pod Namespaces

6 How do we use Chaos Mesh?

For the CERN Kubernetes service we rely on the upstream helm chart with custom definitions and templates for definitions of experiments and some added dependencies. The directory structure looks like this:



6.1 Installation

1. Cluster setup, relying on the existing CERN Kubernetes service:

```
[nprasad@lxplus8s20 ~]$ kubectl get nodes
NAME                                STATUS   ROLES    AGE   VERSION
nprasad-cluster-v4du2n5n53az-master-0   Ready    master   36d   v1.21.1
nprasad-cluster-v4du2n5n53az-node-0     Ready    <none>   36d   v1.21.1
nprasad-cluster-v4du2n5n53az-node-1     Ready    <none>   36d   v1.21.1
[nprasad@lxplus8s20 ~]$
```

2. Add the Chaos Mesh repository.

```
helm repo add chaos-mesh https://charts.chaos-mesh.org
```

3. Create a namespace to install Chaos Mesh

```
kubectl create ns <namespace>
```

4. Install Chaos Mesh adding specific options for containerd

```
helm install chaos-mesh chaos-mesh/chaos-mesh -n=<namespace> --set chaosDaemon.runtime
=containerd --set chaosDaemon.socketPath=/run/containerd/containerd.sock
--set dashboard.securityMode=false --version 2.3.0
```

5. Finally install the CERN specific ChaosMesh templates via our chart

```
helm install chaosmesh-chart chaosmesh/ --values chaosmesh/values.yaml
```

6.2 Chaos Mesh Experiments

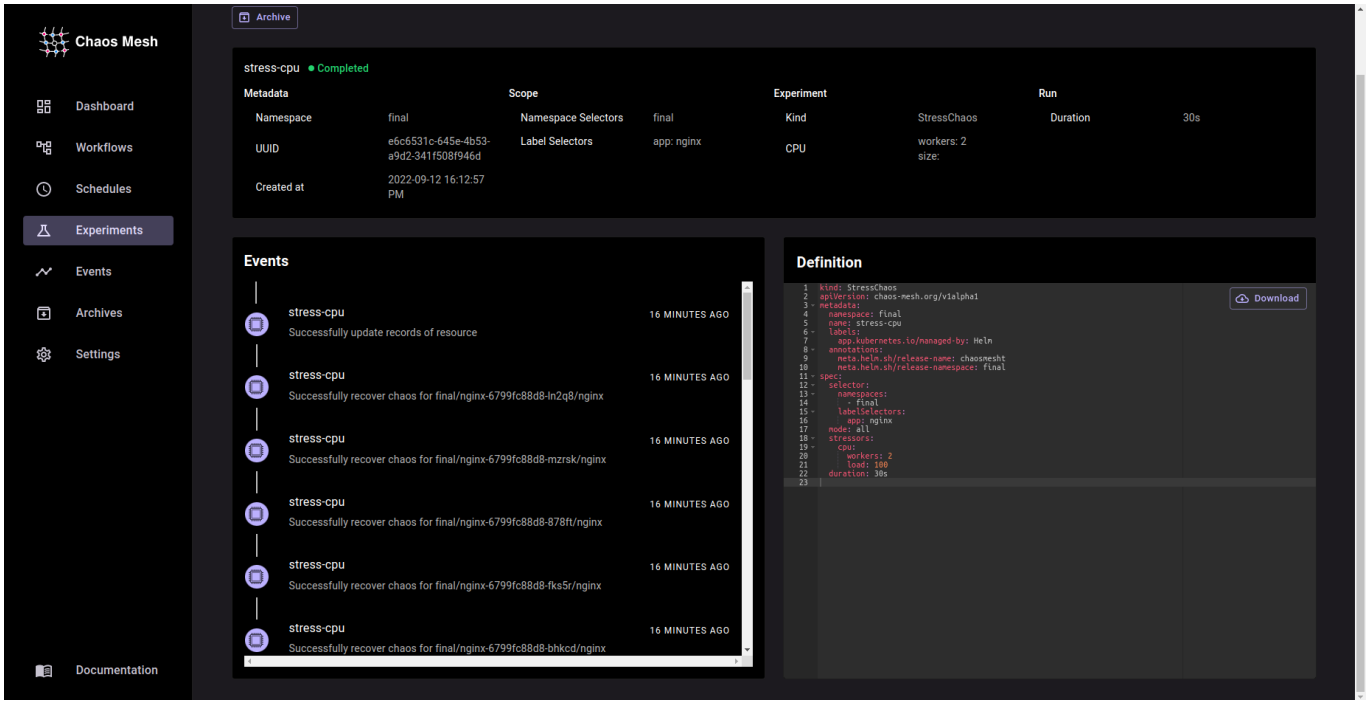
6.2.1 Stress CPU

Let's start with the **stress-cpu.yaml** experiment. The experiment is defined as follows:

```
{{- if .Values.experiments.stressCpu }}
{{- $namespaces := .Values.namespaces }}
{{- range $app := .Values.apps }}
apiVersion: chaos-mesh.org/v1alpha1
kind: StressChaos
metadata:
  name: stress-cpu
  namespace: {{ $.Release.Namespace }}
spec:
  mode: all
  selector:
    namespaces:
      - {{ $namespaces }}
    labelSelectors:
      app: "{{ $app }}"
  stressors:
    cpu:
      workers: 1
      load: 100
  duration: "30s"
{{- end }}
{{- end }}
```

This will add a single CPU stress worker to each of our nginx pods that will load the pods 100% of the scheduled time. These workers will be scheduled and run for 30s in the pod, meaning we should expect to see our Nginx pods' CPU spike for 30s and then drop back to near 0.

For the visualization we can check the dashboard which shows it has been deployed and running.



Monitoring CPU usage can be done via a Prometheus deployment or with a simple command line.

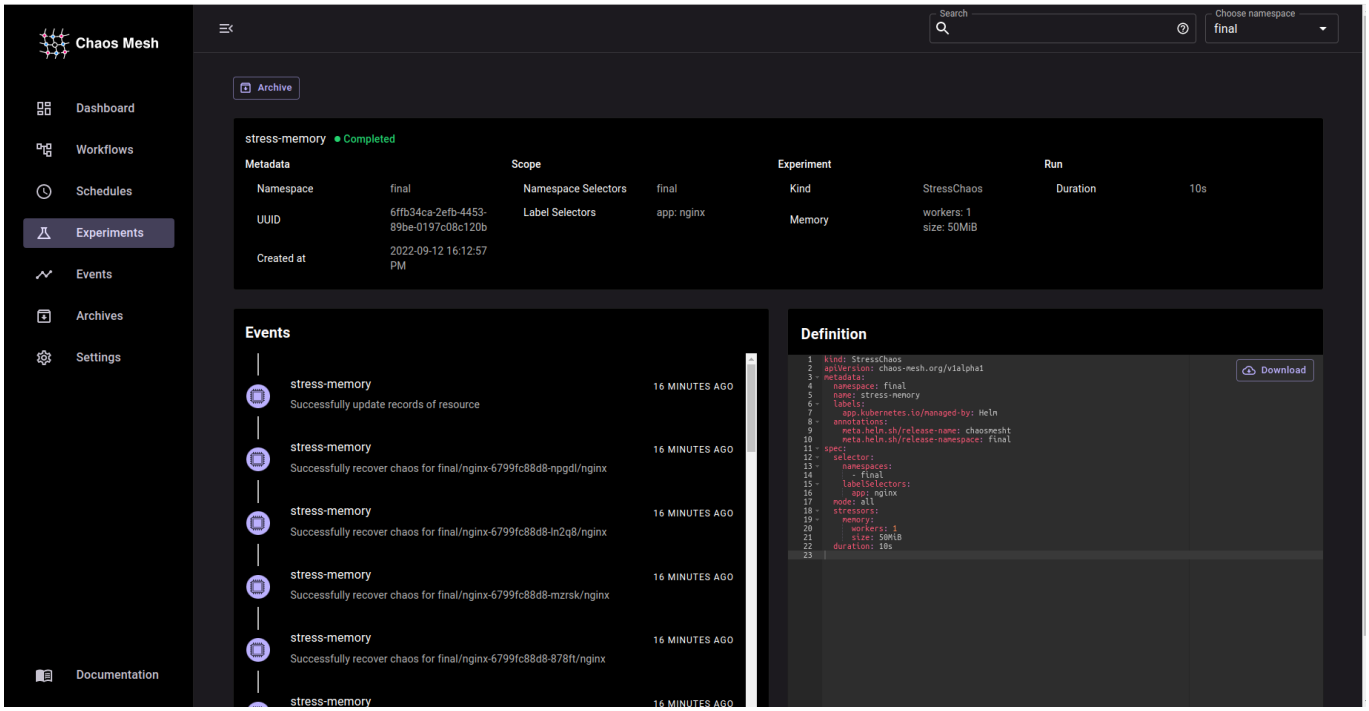
```
watch kubectl top pods -n <namespace>
```

6.2.2 Stress Memory

A similar experiment was configured focusing on memory. In this case we add artificial memory usage to all nginx pods and keep that state for 10 seconds.

```
{{- if .Values.experiments.stressMemory }}
{{- $namespaces := .Values.namespaces }}
{{- range $app := .Values.apps }}
apiVersion: chaos-mesh.org/v1alpha1
kind: StressChaos
metadata:
  name: stress-memory
  namespace: {{ $.Release.Namespace }}
spec:
  mode: all
  selector:
    namespaces:
      - {{ $namespaces }}
    labelSelectors:
      app: "{{ $app }}"
  stressors:
    memory:
      workers: 1
      size: 50MiB
  duration: "10s"
{{- end }}
{{- end }}
```

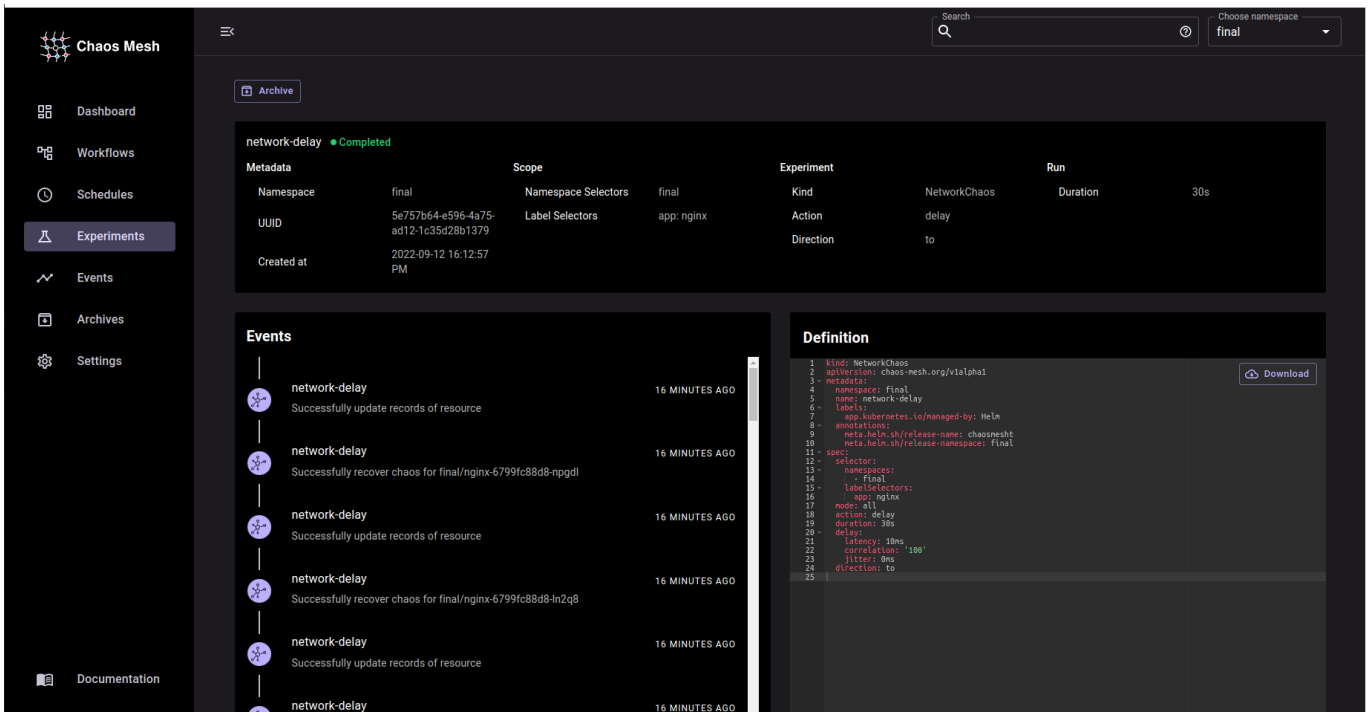
As before, we can check the dashboard to check the experiment definition and runs.



6.2.3 Network Delay

We define an experiment causing a latency of 10 milliseconds in the network connection of the target pods during 30 seconds.

```
{{- if .Values.experiments.networkDelay }}
{{- $namespaces := .Values.namespaces }}
{{- range $app := .Values.apps }}
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: network-delay
  namespace: {{ $.Release.Namespace }}
spec:
  action: delay
  mode: all
  duration: '30s'
  selector:
    namespaces:
      - {{ $namespaces }}
    labelSelectors:
      app: "{{ $app }}"
  delay:
    latency: '10ms' #indicates the network latency
    correlation: '100' #correlation b/w the current latency and previous one
    jitter: '0ms' #indicates the range of the network policy
  direction: to
{{- end }}
{{- end }}
```



We can again rely on the Prometheus monitoring or simply check the induced latency with a ping to the nginx pods.

```
ping <IP of the Nginx Pod> -c 2
```

6.2.4 Workflow

Finally we have implemented tests for Pod kill and Pod failure via a workflow. Now you'll wonder why we didn't perform these experiments like others. So, before explaining the reason let me introduce the term workflow.

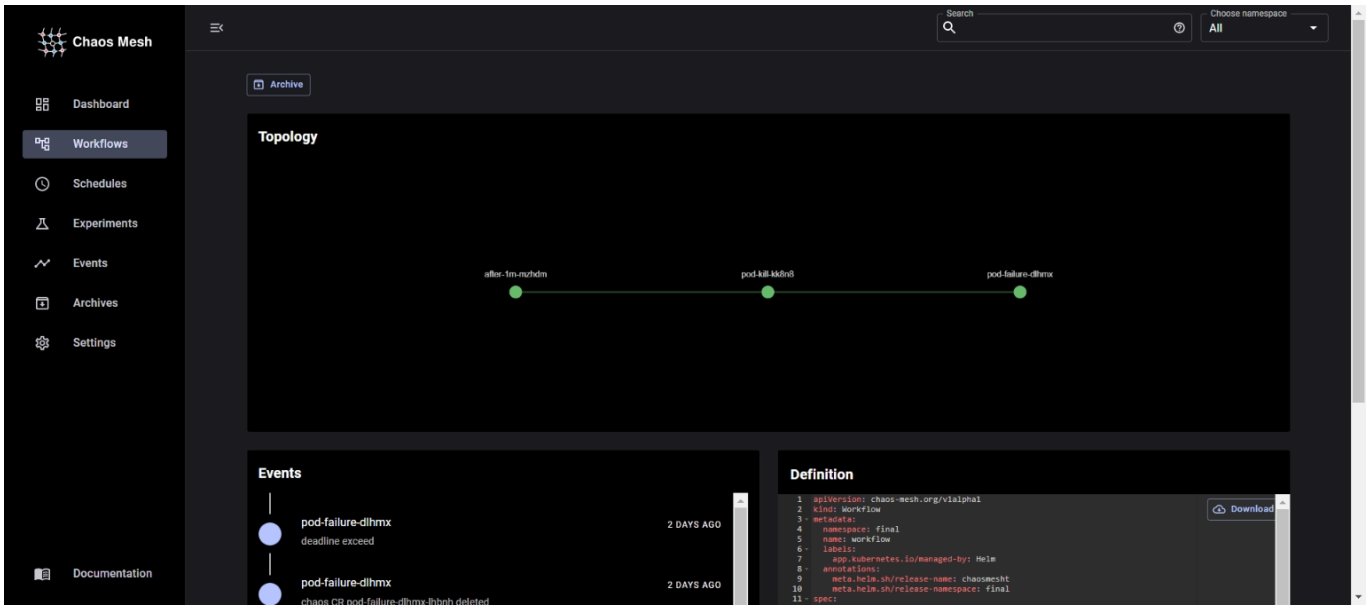
What is WorkFlow?

Simulating real system faults using Chaos Mesh implies continuous validation and might require building a series of concurrent faults instead of performing individual Chaos injections.

To meet this need Chaos Mesh provides Workflows and a corresponding workflow engine. With this engine different Chaos experiments can be run in series or parallel to simulate production-level errors.

Why Workflow?

Pod kill and pod failure make the pod unavailable for a certain period of time, making it unsuitable to be run along other experiments - additional experiments will be unable to find the target pods. With workflows it is possible to delay pod-kill and pod-failure experiments for a certain period of time, allowing other experiments to complete their work.



7 Daemonset

In addition to the experiment templates the published helm chart also includes a definition of a DaemonSet, which covers the need to preload the NET_SCH_NETEM module installed in every node.
The module loading script is included in a ConfigMap which is run on every node through the DaemonSet.

8 Conclusions

Through this work we have selected and validated ChaosMesh as a valid option for CERN users to deploy and configure different kinds of chaos experiments, as well as monitoring those experiments via the ChaosMesh dashboards.

We have added the option to deploy these configurations in all CERN Kubernetes clusters, via a helm chart which includes all dependencies and a set of pre-defined experiments. The main goal being to reduce the effort for a wide range of applications to enable chaos experiments in their setups. Next steps should be to involve more users of the service to try it out and expand the number of pre-configurations being offered by the standard CERN Kubernetes service deployments.